Designing for Hostile Environments

Monte recently designed a CPU that will one day orbit Jupiter in one of the most hostile environments in the solar system. In this article, he describes design techniques that you can use to prepare your systems for operation in hostile environments.

I recently had the opportunity to design a CPU that will be used to control a science instrument on NASA's Juno mission to Jupiter, which is scheduled to launch in 2010. Orbiting in one of the most hostile environments in the solar system places some special requirements on the design, but those requirements are not unique to that situation.

In this article, I'll go over the requirements for the design, some of the design techniques I used, and a number of the trade-offs I had to make. While your next design might not be traveling as far as mine will, many of these design considerations can be used to make any design more robust.

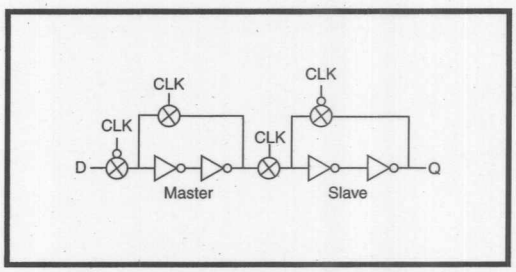

Figure 1—A flip-flop is the combination of two latches, each controlled by the opposite phase of the clock.

THE PROBLEM

The basic problem goes by many names: soft error, alpha-induced error, neutron-flux error, ionizing-radiation induced upset, transient radiation upset, and single-event upset (SEU). I'll stick with SEU because it doesn't specify the source of the error, just the result.

Implicit in the name, single-event upset is the idea that the effect of an error starts on a single circuit node. This is an important assumption, because even though an error may propagate throughout the design via the normal circuit paths, the root of the error can be traced to a single node. Without this simplifying assumption, hardening a design against errors is almost impossible.

Unfortunately, as technology advances, this basic assumption will become less and less accurate because disparate circuit elements will be close enough together for an errant particle to affect more than one circuit node. But we're not there yet.

In a purely combinatorial design with no feedback paths, an SEU transiently flips the state of a single node in the circuit. This state change may propagate throughout the circuit, but the driver on that node will quickly force the node back to the correct state and the error will persist only for a few nanoseconds. But once there is feedback in the circuit, in the form of latches or flip-flops, an SEU can cause

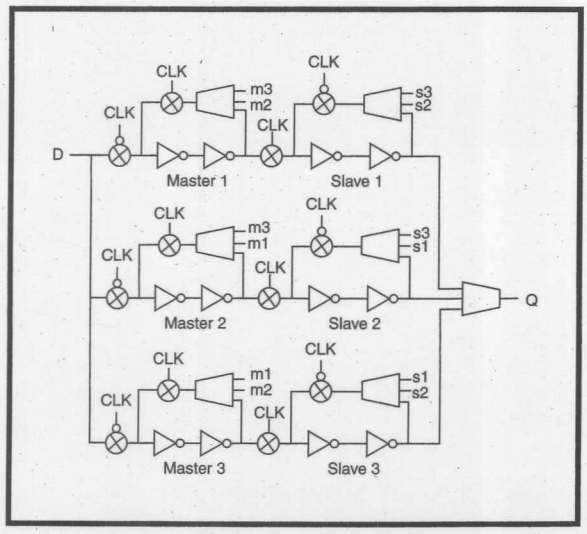

Figure 2—Hardening the flip-flop with triple-modular redundancy makes each flip-flop much more complex.

Input A	Input B	Input C	Output
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Table 1—The two-of-three voter circuit protects against one input being in the wrong state.

a lot of problems. To see why this is the case, refer to the simplified schematic diagram of a flip-flop in Figure 1.

Each half of a flip-flop is a latch that holds its state during one half of a clock cycle. If one of the nodes in a latch is disturbed while it is storing a value, the state of the latch will flip and the feedback will hold the disturbed state until the clock refreshes the latch.

If the slave (output) half of the flip-flop is disturbed while it is in the hold state, the error will propagate throughout the network connected to the output and will be sampled on the next clock edge by subsequent flip-flops. If the master (input) half of the flip-flop is disturbed while it is in the hold state, the error will propagate through the slave half of the flip-flop and be sampled by subsequent flip-flops. Thus, flip-flops are vulnerable throughout the clock cycle.

Data	Check 2 (b3 ⊕ b2 ⊕ b1)	Check 1 (b3 ⊕ b2 ⊕ b0)	Check 0 (b3 ⊕ b1 ⊕ b0)	Result
b3 b2 b1 b0	g2	g1	g0	b3 b2 b1 g2 b0 g1 g0
0000	0	0	0	0000000
0001	0	1	1	0000111
0010	1	0	1	0011001
0011	1	1	0	0011110
0100	1	1	0	0101010
0101	1	0	1	0101101
0110	0	1	1	0110011
0111	0	0	0	0110100
1000	1	1	1	1001011
1001	1	0	0	1001100
1010	0	1	0	1010010
1011	0	0	1	1010101
1100	0	0	1	1100001
1101	0	1	0	1100110
1110	1	0	0	1111000
1111	1	1	1	1111111

Table 2—This example of a Hamming code adds three generated check bits to each of the 4 data bits. The check bits cover different combinations of data bits and are input to the flip-flops.

FIRST LEVEL OF DEFENSE

The first level of defense against an SEU involves making the flip-flops impervious to upsets. The most common way of doing this involves triple-modular redundancy (TMR), as shown in Figure 2.

Basically, each individual latch in a flip-flop is replaced by three latches. The feedback path for each of the latches is replaced by a "voting" circuit. Each independent voting circuit outputs the majority value of its three inputs (see Table 1). If each latch and voting block is separated sufficiently to guarantee that only one can be upset by an event, the output of the flip-flop will be tolerant of an SEU.

It should be obvious that TMR can be expensive because each hardened flip-flop is more than three times as complicated as a basic flip-flop. Even more complex schemes are possible, starting with three-of-five voting techniques. But more complexity leads to diminishing returns because the added complexity also increases the circuit area and thus the cross-sectional area vulnerable to an upset.

Implementing the TMR circuitry for every flip-flop in a design by hand would be a lot of work; fortunately, the target device for the CPU is specifically designed for this type of application. The Actel RTAX-S family of FPGAs implements TMR for every flip-flop in the device (transparently to the user). According to Actel, the layout of these flip-flops is such that the SEU model is applicable.

At this point, you might think that we've done everything necessary to harden a design, but in fact we've just begun. The remaining thorny problem occurs when an SEU happens when there is a clock edge. If a flip-flop samples an incorrect value, none of the TMR circuitry inside the flip-flop is going to correct the error.

Admittedly, this circumstance is unlikely because the sampling window for a flip-flop is extremely short, but it can happen. The more hostile the environment, the more likely it is. So we need to take steps to make the design tolerant of these types of errors.

TO CORRECT OR NOT TO CORRECT

One technique that can be used to handle errors that might make it through the TMR protection is to use a single-error-correcting (SEC) code for collections of flip-flops. In fact, the technique can be used in place of TMR.

One of the most robust ways to do this is to add Hamming error-check bits to every collection of flip-flops. The SEC code uses a number of parity bits, which each checks different combinations of data bits in such a way that the resulting parity automatically indicates the position of an error.

Bit arrangement	Check 2 (b3 ⊕ b2 ⊕ b1 ⊕ g2)	Check 1 (b3 ⊕ b2 ⊕ b0 ⊕ g1)	Check 0 (b3 ⊕ b1 ⊕ b0 ⊕ g0)
b3 b2 b1 g2 b0 g1 g0	c2	c1	c1
0000000	0	0	0
1000000	1	1	1
0100000	1	1	0
0010000	1	0	1
0001000	1	0	0
0000100	0	1	1
0000010	0	1	0
0000001	0	0	1

Table 3—The flip-flop outputs generate a check syndrome, which automatically flags which bit is in error and needs to be toggled.

State	Meaning	Encoding	Default next
TR	Reset state	000	T1
T1	First clock of machine cycle	001	T2
TW	Wait state	010	T3
T2	Second clock of machine cycle	011	T3
T4	Single-clock machine cycle	100	T1
T3	Third clock of machine cycle	101	T1
TX	Bus acknowledge (from T3)	110	T1
TY	Bus acknowledge (from T4)	111	T4

Table 4—The clock_cyc state machine has no unused states, but TW, TX, and TY can persist forever based on external inputs.

Table 2 shows an example of the Hamming code for 4 bits of data. Each generated check bit is the result of an even parity check of three of the data bits. The generated check bits are then latched in flip-flops with the data in the order shown in the table.

Table 3 shows what happens on the output side of the flip-flops, with and without an error bit. On this side of the flip-flops, each of the three check bits is generated from the even parity of 4 bits. The check bits then directly indicate which input bit is in error and should be toggled to create the original state.

If you have been reading carefully, you should notice that TMR is equivalent to a Hamming error-check code for 1 bit of data. That is why the two techniques are rarely used together. While an SEC code will correct one error among each group of flip-flops, TMR protects each flip-flop individually, so it is more robust overall.

In this design, because the underlying technology employs TMR, the specification did not require the use of an SEC code for flip-flops in the design. Instead, my customer required that I guarantee only that the design could not get into an unrecoverable state in response to a single bit error. This is understandable given the finite resources present in the target FPGA.

STATE MACHINES

Every flip-flop or collection of flip-flops is a state machine. A CPU can be viewed as either a large state machine with a huge number of states or a collection of smaller interconnected state machines. For our purposes, it's easier to view a CPU as a collection of small individual state machines.

The effect of an SEU on a state machine is that one bit of the state will toggle. Because I am not correcting errors, the response of a state machine to this potentially illegal transition is critical to tolerating an upset. For this discussion, state machines can be divided into three categories.

The first type of state machine is what I'll call user-controlled registers. They are the usual CPU registers that are visible to (and controlled directly by) the program and the internal temporary registers that are used during instruction execution to hold data, intermediate results, and addresses.

All of the registers can store any data pattern. As a result, they do not have any illegal states. Upsets to those types of state machines will usually be manifest only as erroneous data or improper operation sequences and are beyond the scope of this discussion. Instead, either software self-checks or circuitry, such as watch-dog timers, must be used as a last line of defense.

I'll call the second type of state machine continuously clocked flip-flops. They are just what the name implies: flip-flops that are updated during every clock cycle. In this CPU design, that type of state machine is used to synchronize input signals, delay (and synchronize) a decoded signal, or create a signal clocked by the opposite edge of the clock.

The response of the flip-flops to a single-event upset will persist only for one clock cycle and will usually create a condition that will be visible on the external signals of the device, so there isn't much more that can be done in the design here either.

The third type of state machine is the classic control type of state machine. It may not use every possible state, so this is where we must pay the most attention in the design. It's imperative that we are able to detect and flag any illegal state and that every possible state has a defined exit path.

State	Meaning	Encoding	Default next
IF1	Instruction fetch—first byte	000001	OF1
DLY	Delay—one clock for DJNZ	000010	OF1
IF2	Instruction fetch—second byte	000100	TRAP
OF1	Operand fetch—first byte	000111	OF2
RD1	Data read—first byte	001000	RD2
RD2	Data read—second byte	001011	IF1
OF2	Operand fetch—second byte	001101	IF1
IOP1	Internal operation—one clock	001110	IF1
WR1	Data write—first byte	010101	WR2
WR2	Data write—second byte	010110	IF1
IF3	Instruction fetch—third byte	011000	RD2
FIOP3	Internal operation—one clock	011001	FIOP2
FIOP2	Internal operation—one clock	011010	FIOP1
FIOP1	Internal operation—one clock	011011	IF1
RRST	Reset default state	001100	IF1
IOP10	Internal operation—one clock	011101	IOP9
IOP9	Internal operation—one clock	100000	IOP8
IOP8	Internal operation—one clock	100001	IOP7
IOP7	Internal operation—one clock	100010	IOP6
IOP6	Internal operation—one clock	100011	IOP5
IOP5	Internal operation—one clock	100100	IOP4
IOP4	Internal operation—one clock	100101	IOP3
IOP3	Internal operation—one clock	100110	IOP2
IOP2	Internal operation—one clock	100111	IOP1
SIOP5	Internal operation—one clock	101001	SIOP4
SIOP4	Internal operation—one clock	101010	SIOP3
SIOP3	Internal operation—one clock	101011	SIOP2
SIOP2	Internal operation—one clock	101100	SIOP1
SIOP1	Internal operation—one clock	101101	WR2
TRAP	Trap acknowledge	110000	SIOP5
INTA	Interrupt acknowledge	111000	WOF1
NMIA	NMI Acknowledge	111001	SIOP2
FAULT	Fault detect	111010	IF1
HLT	Halt	111110	HLT
SLP	Sleep	111111	SLP
	Any unused state		FAULT

Table 5—Only 35 out of the 64 possible states are valid for the mach_cyc state machine, but every possible state must have an exit path.

Detecting and flagging illegal states is easy. All that is required is a decoder and an output signal that can be used by external error-recovery logic. In this application, the fault detect output will probably force a nonmaskable interrupt to guarantee a restart from a known state.

Providing a defined exit path for every possible state is something that most designers don't think about because it takes extra logic. Besides, the logic can't be tested because there is no normal way to get into an illegal state. But, in this application, I can't afford to have an illegal state that, once entered, cannot be exited. This is where I had to spend much of my design time.

DESIGN DETAILS

The CPU that I was asked to design for this application is essentially a copy of the venerable Zilog Z180. It's a basic 8-bit processor that is an upgrade to the original Z80 CPU. For brevity, I won't go over the details of the instruction set or register architecture here. Instead, I will concentrate on the features that are important to this discussion.

The CPU executes instructions using machine cycles that vary in length from one to three clock cycles. As a result, there are two main control-type state machines in this design: clock_cyc and mach_cyc.

The clock_cyc state machine is 3 bits wide and tracks the clock cycle within a machine cycle. Fortunately, there are no unused states in this state machine, although the default (reset) state is used only while in reset.

Table 4 shows the states and default sequence for this state machine. Because there is a specific sequence of clock cycles for each machine cycle, any illegal state transition in clock_cyc should result in improper instruction execution and illegal combinations of external output signals.

The clock_cyc state is always updated on every clock cycle so you normally don't need to worry about errors persisting. However, there are two cases where this state machine can fail to advance to the default case. This happens if the WAIT input or BUSREQ input is held active.

Either case is valid as far as the CPU is concerned, so there is nothing you can do here in hardware. But, this is a case where the design of the overall system must prevent this from ever happening, usually by building in some kind of time limit on the amount of time that the two signals can be active.

The mach_cyc state machine is 6 bits wide and controls the sequence of bus cycles and internal operation cycles for each instruction. There are 34 valid states required for this state machine, leaving 30 states unused.

Table 5 shows the information for the mach_cyc state machine. I spent much of my design time assigning the encoding for the valid states because there are a number of potential problems with this state machine.

The biggest potential problem with the mach_cyc state machine is that two of the required states can naturally persist for a long time. Both the

Halt and Sleep states, by definition, will persist until either an interrupt or reset occurs. As a result, it is imperative that the encoding of the states not allow any other valid state to transition to either of the states as a result of an upset.

A second potential problem is that there are block move instructions that use a 16-bit CPU register to count the length of the block transfer. This can tie up the processor for an inordinate amount of time, so the special states that are required for block moves should be protected in the same way, if possible.

The final consideration for encoding the mach_cyc states is the issue of frequency. It makes sense to try to guarantee that illegal transitions will be detected from those states that occur most often, while leaving infrequently used states less protected.

Table 6 shows all of the states for the mach_cyc state machine along with the corresponding state that results from a single bit flip. I put all of those unused states to good use. All of the blank entries in this table correspond to an unused and illegal state.

The next state for any unused state will always be the Fault state, which activates the chip output of the same name to signal to external circuitry that an illegal state has been detected. The Fault state lasts just one clock cycle and then the CPU resumes fetching instructions.

Going back to the potential problems with this state machine, you'll notice that both the Halt and Sleep states can never be reached because of an error while in any other valid state. The fact that an upset can cause a transition between the two states is not an issue. The states are essentially identical anyway, and the only difference between them is whether the Halt or Sleep signal out of the CPU is active.

The block-move instructions use the FIOP states to decide whether or not the block is finished, so illegal transitions to the states have been minimized. There aren't enough unused states to completely eliminate illegal transitions into the states.

The end result isn't too bad though, because the instructions are actually re-fetched for each iteration in this design. If this were not the case, I would have had to eliminate all possible illegal transitions into the states.

As far as the final consideration, I think that I've reached a reasonable compromise. All of the main states have only one or two undetected illegal transitions. The only exception is the IF3 state, but that is used only for the rare 3-byte opcode instructions anyway.

All of the internal operation states are only one clock cycle long, so they are inherently less susceptible to upset than three-clock-cycle bus cycle states. In addition, some of them are used only for a few instructions. For example, the IOP2–IOP10 states are used only by the multiply instruction. The IOP1 state is used by many instructions, but it is encoded so any upset will be detected.

State	Encoding	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
IF1	000001	IOP8	—	—	—	—	—
DLY	000010	IOP7	—	—	—	—	—
IF2	000100	IOP5	—	—	—	—	—
OF1	000111	IOP2	—	—	—	—	—
RD1	001000	—	IF3	—	—	—	—
RD2	001011	SIOP3	FIOP1	—	—	—	—
OF2	001101	SIOP1	IOP10	—	—	—	—
IOP1	001110	—	—	—	—	—	—
WR1	010101	—	—	IOP10	—	—	—
WR2	010110	—	—	—	—	—	—
IF3	011000	INTA	RD1	—	RRST	FIOP2	FIOP3
FIOP3	011001	NMIA	—	—	IOP10	FIOP1	IF3
FIOP2	011010	—	—	—	—	IF3	FIOP1
FIOP1	011011	—	RD2	—	—	FIOP3	FIOP2
RRST	001100	—	—	—	—	—	—
IOP10	011101	—	OF2	—	FIOP3	—	RRST
IOP9	100000	RRST	TRAP	—	IOP5	IOP7	IOP8
IOP8	100001	IF1	—	SIOP5	IOP4	IOP6	IOP9
IOP7	100010	DLY	—	SIOP4	IOP3	IOP9	IOP6
IOP6	100011	—	—	SIOP3	IOP2	IOP8	IOP7
IOP5	100100	IF2	—	SIOP2	IOP9	IOP3	IOP4
IOP4	100101	—	—	SIOP1	IOP8	IOP2	IOP5
IOP3	100110	—	—	—	IOP7	IOP5	IOP2
IOP2	100111	OF1	—	—	IOP6	IOP4	IOP3
SIOP5	101001	—	NMIA	IOP8	SIOP1	SIOP3	—
SIOP4	101010	—	—	IOP7	—	—	SIOP3
SIOP3	101011	RD2	—	IOP6	—	SIOP5	SIOP4
SIOP2	101100	—	—	IOP5	—	—	SIOP1
SIOP1	101101	OF2	—	IOP4	SIOP5	—	SIOP2
TRAP	110000	—	IOP9	INTA	—	—	—
INTA	111000	IF3	—	TRAP	—	—	NMIA
NMIA	111001	FIOP3	SIOP5	—	—	—	INTA
FAULT	111010	—	—	—	—	—	—
HLT	111110	—	—	—	—	—	SLP
SLP	111111	—	—	—	—	—	HLT

Table 6—All of the possible error cases for mach_cyc must be looked at. All of the possibilities marked "—" are guaranteed to be detected by the hardware.

WILL IT WORK?

It's one thing to include all of the fault detection in the design, but one of the customer requirements was for me to verify that it all worked the way it was supposed to. Trying to do this in hardware in an FPGA could be very time-consuming, but doing it in simulation is pretty simple.

The first step was to verify that the design worked properly in the absence

Listing 1—Verilog makes it easy to reach down into a design hierarchy to simulate the effect of an upset.

```
/*****************************************************************/
/*                                                               */
/* fault injection                                               */
/*                                                               */
/*****************************************************************/
assign FAULT_MACH = 6'b000001;          /* fault pattern match   */
assign FORCE_MACH = 6'b000000;          /* fault forced value    */

initial begin
    wait(Y180.mach_cyc == FAULT_MACH); /* wait for state         */
    #25 FORCE_EVENT <= 1'b1;
    #25 FORCE_EVENT <= 1'b0;
    end

always @ (posedge FORCE_EVENT) Y180.M_STATE.mach_cyc <= FORCE_MACH;
```

of an upset. This required a test suite that exercised every instruction, every flag combination, and enough data combinations to verify all of the logic paths. The resulting test suite exercises the design for just over 360,000 clock cycles in the simulator, or the equivalent of 36 ms of real time.

Once the test suite was complete, Verilog made it easy to inject upsets into the design by forcing the state of a flip-flop to toggle. The Verilog "reg-er" data type, which is used to model storage elements like flip-flops, automatically holds the state once triggered. So all that is required to inject an error is to add a triggering event to the desired flip-flop.

Listing 1 shows the code I used in the test bench to simulate an upset. The `Y180.mach_cyc` variable is the hierarchical name of the mach_cyc state machine. The code first waits for a particular state to occur in the mach_cyc state machine and then activates a test bench signal called `FORCE_EVENT`. The rising edge of the test bench signal is then used as a trigger to force a different value into the mach_cyc state machine.

Of course, it is possible to add more complex conditions to the creation of the triggering signal. For example, I could add a line to wait for a certain amount of time before checking for the state to fault, or a line to wait for a specific instruction to be executed, or multiple occurrences of any of these conditions. But that is the basic idea.

At this point, it was simply a matter of running multiple simulations to verify that every possible illegal state was properly flagged with a Fault signal. With the relatively small number of cases to be checked, I did them individually, checking visually for the correct response. It would also be possible to automate the task and have the test bench check for the Fault signal.

In the final system design, the Fault signal will probably be used to trigger a nonmaskable interrupt to attempt to recover from the error. Events that activate the Fault output will not have corrupted any of the data in the CPU registers, but the instruction in progress at the time may or may not have completed. This makes recovery a somewhat difficult proposition.

Upsets that cause transitions to a valid state will not activate the Fault signal and may corrupt the data in the CPU registers. So, fault-checking mechanisms must be employed in the software to detect those cases.

WRAPPING UP

I could have made the state machines even more impervious to upsets by adding another bit to the mach_cyc state machine and taking advantage of the increased number of illegal states. But that would have required more logic, which was a problem given the fixed resources of the target FPGA. So this will have to do.

Like most designers, I never paid much attention to the unused states in state machines and I rarely bothered to make sure that unused states wouldn't persist forever. During the design process, it is very handy to make the default (unused) states always go unknown in a simulation because it is an easy way to quickly find design errors.

But once the design is mostly debugged, it doesn't hurt to put in a defined exit path for every state into the design. Yes, it adds a few gates, but this experience has shown me that it never hurts to be safe.

Monte Dalrymple (monted@systemyde.com) has been designing integrated circuits for 28 years. Since tiring of the corporate world, he has been designing on a contract basis. Monte is the designer of all four generations of Rabbit microprocessors.

RESOURCES

Actel Corp., "RTAX-S/SL RadTolerant FPGAs," 2007, www.actel.com/documents/RTAXSGenDesc_DS.pdf.

Answers Corp., "Radiation hardening," 2007, www.answers.com/topic/radiation-hardening.

R. W. Hamming, "Error Detecting and Error Correcting Codes," *Bell System Technical Journal*, Vol. 29, 1950.

National Aeronautics and Space Administration, "Juno mission overview," 2007, http://newfrontiers.nasa.gov/missions_juno.html.

Systemyde International Corp., "Y180-S: 8-Bit Microprocessor Synthesizable Verilog HDL Model User Manual," 2006, www.systemyde.com/pdf/y180s.pdf.

SOURCES

RTAX-S/SL RadTolerant FPGAs
Actel Corp.
www.actel.com

Z180 Microcontroller
Zilog, Inc.
www.zilog.com